Detection of illegal control flow in Android System: Protecting private data used by Smartphone Apps

Mariem Graa, Nora Cuppens-Bouhlahia, Frédéric Cuppens, Ana Cavalli

Detection of illegal control flow in Android System: Protecting private data used by Smartphone Apps

Mariem Graa[1,2], Nora Cuppens-Boulahia[1], Frédéric Cuppens[1], Ana Cavalli[2]

[1]Telecom-Bretagne, 2 Rue de la Châtaigneraie, 35576 Cesson Sévigné - France
{mariem.benabdallah,nora.cuppens,frederic.cuppens}@telecom-bretagne.eu
[2]Telecom-SudParis, 9 Rue Charles Fourier, 91000 Evry - France
{mariem.graa,ana.cavalli}@it-sudparis.eu

Abstract. Today, security is a requirement for smartphone operating systems that are used to store and handle sensitive information. However, smartphone users usually download third-party applications that can leak personal data without user authorization. For this reason, the dynamic taint analysis mechanism is used to control the manipulation of private data by third-party apps [9]. But this technique does not detect control flows. In particular, untrusted applications can circumvent Android system and get privacy sensitive information through control flows. In this paper, we propose a hybrid approach that combines static and dynamic analysis to propagate taint along control dependencies in Android system. To evaluate the effectiveness of our approach, we analyse 27 free Android applications. We found that 14 of these applications use control flows to transfer sensitive data. We successfully detect that 8 of them leaked private information. Our approach creates 19% performance overhead that is due to the propagation of taint in the control flow. By using our approach, it becomes possible to detect leakage of personal data through control flows.

Key words: Android system; Smartphones; Dynamic analysis; Static analysis; Control dependencies; Leakage of sensitive information

1 Introduction

Smartphone operating systems use has been increasing at an accelerated rate in recent years. Android surpassed 80% market share in the third quarter of 2013 [17] and it is the most targeted OS by the cyber criminals with more than 98% of malware applications [9]. This is due to the prevalence of third party app stores (48 billion apps have been installed from the Google Play store in May 2013[19]). These applications are used to capture, store, manipulate, and access to data of a sensitive nature in mobile phone. An attacker can launch control flow attacks to compromise confidentiality of the Android system and can leak private information without user authorization. In the study presented in the

Black Hat conference, Daswani [21] analyzed the live behavior of 10,000 Android applications and showed that more than 800 of them were found to be leaking personal data to an unauthorized server. Therefore, there is a need to provide adequate security mechanisms to control the manipulation of private data by third-party apps. Many mechanisms are used to protect sensitive data in the Android system, such as the dynamic taint analysis that is implemented in TaintDroid [9].

```
1.boolean b = false;
2.boolean c = false;
3.if (!a)
4. c = true;
5.if (!c)
6.  b = true;
```

Fig. 1. Implicit flow example.

The principle of dynamic taint analysis is to "taint" some of the data in a system and then propagate the taint to data for tracking the information flow in the program. Two types of flows are defined: explicit flows such as $x = y$, where we observe an explicit transfer of a value from x to y, and implicit flows (control flows) shown in Figure 1 were there is no direct transfer of value from a to b, but when the code is executed, b would obtain the value of a. The dynamic taint analysis mechanism does not detect control flows which can cause an under tainting problem *i.e.* that some values should be marked as tainted, but are not. The under tainting problem can cause a failure to detect a leak of sensitive information. Thus, malicious applications can bypass the Android system and get privacy sensitive information through control flows. In a previous work [12], we have proposed an approach that combines static and dynamic taint analysis to propagate taint along control dependencies and to track implicit flows in the Google Android operating system. Our approach enhances the TaintDroid approach by tracking control flows in the Android system to solve the under-tainting problem. In this paper, we present implementation details and experimental results of the proposed approach. We show effectiveness of our approach to propagate taint in the conditional structures of real Android applications and to detect leakage of sensitive information. This paper is organized as follows: Section 2 presents a motivating example. We discuss related work about static and dynamic taint analysis and we analyze existing solutions to solve the under tainting problem in Section 3. Section 4 describes the proposed approach and the corresponding implementation details. We analyse a number of Android applications to test the effectiveness of our approach and we study our approach taint tracking overhead in Section 5. Finally, Section 6 concludes with an outline of future work.

2 Motivating example

An attacker can exploit an indirect control dependencies to leak private data. Let us consider the control dependence attack shown in Figure 2. The variable X contains the private data that is the user contact. The

```
String X = contact_name;
String Y="";
char[] TabAsc;
 int k=0;
TabAsc = new char [96];

while (codeAsc < 0x80) {

  for (column = 0; column < 16; column++) {
     TabAsc[k] = codeAsc;
    codeAsc++;
    k++;
  }
  row++;
}

for (int i = 0;i< X.length(); i++)
{
    char x=X.charAt(i);

    for (int j=1;j<TabAsc.length;j++)
    {
         if (x==TabAsc[j])
         Y=Y+TabAsc[j];
    }

}

NetworkTransfer(Y);
```

Fig. 2. Control dependence attack

attacker tries to get the user contact name by comparing it with symboles of Ascii table in the second loop. He stored the character of private data founded in Y. At the end of the loop, the variable Y contains the correct value of the user contact and it is not tainted because taint is not propagated in the control flow statement. The attacker exploits untainted variable that should be tainted (under tainting problem) to leak private data. Thus, Y is leaked through the network without being detected. Therefore, an attacker can leak a sensitive information by exploiting control flows.

3 Related Work

Many security mechanisms are used to protect sensitive data in smartphones. TaintDroid [9], an extension of the Android mobile-phone used to control in realtime the manipulation of users personal data by third-party applications. It implements a dynamic taint tracking and analysis system to track the information flow and to detect when sensitive data leaves the system. AppFence [15] extends Taintdroid to implement enforcement

policies. One limit of TaintDroid and AppFence approaches is that they cannot propagate taint in control dependencies. The methods proposed in [8, 5, 11] statically analyze third party application code for detecting data leaks. But, these static analyses approaches cannot capture all runtime configuration. Some approaches combine static and dynamic analysis to solve the under-tainting problem. BitBlaze [20] presents a novel fusion of static and dynamic taint analysis techniques to track all information flow. DTA++ [16] uses Bitblaze and enhances the dynamic taint analysis to limit the under-tainting problem. However DTA++ is evaluated only on benign applications. Trishul [18] correctly identifies control flow to detect a leak of sensitive information. Furthermore, these approaches are not implemented in smartphones application. Fenton [10] defined a Data Mark Machine, an abstract model, to handle control flows. This model does not take into account the implicit flow when the branch is not executed. Denning [7] enhances the run time mechanism used by Fenton with a compile time mechanism to solve the under-tainting problem. Denning inserts updating instructions whether the branch is taken or not. We draw our inspiration from the Denning approach, but we define formally a set of taint propagation rules to solve the under-tainting problem. In [12], we propose a hybrid approach that tracks control flows in smartphones. We define a set of formal propagation rules to solve the under-tainting problem. We prove the correctness and completeness of these rules and we propose a correct and complete algorithm to solve the under tainting problem [13]. In [14], we show that our approach can resist to code obfuscation attacks based on control dependencies in the Android system using the taint propagation rules. But, we do not evaluate the overhead and effectiveness of our approach and do not test a real Android applications. We provide test exprimental results and we evaluate the overhead and the false positives of our approach. We show that it successfully detects sensitive information leakage by untrusted Android applications.

4 Approach Overview and Implementation

Our objective is to detect private information leakage by untrusted smartphone applications exploiting implicit flows. We control the manipulation of private data by third party application in realtime.

Our approach consists of two main components: the *StaticAnalysis* component and the *DynamicAnalysis* component (see Figure 3). We implement our proposed approach in the TaintDroid operating system. We add a *StaticAnalysis* component in the Dalvik virtual machine verifier that statically analyzes instructions of third party application Dex code at load time. Also, we modify the Dalvik virtual machine interpreter to integrate the *DynamicAnalysis* component. We implement the two additional rules using native methods that define the taint propagation.

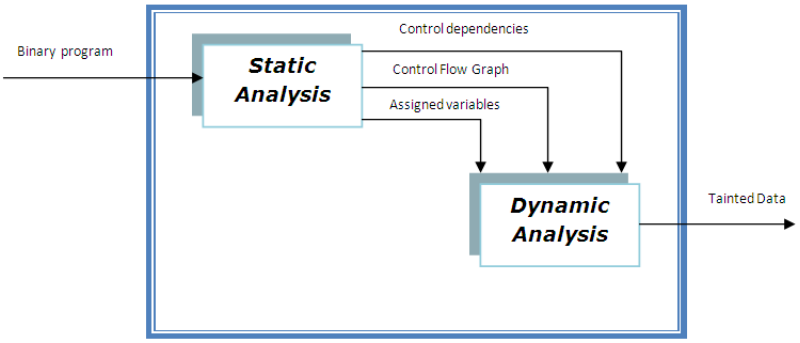

Fig. 3. Our Approach Architecture

4.1 Static Analysis Component

In this component, we check the instructions of methods to create the control flow graph (CFG). A CFG is composed of basic blocks and edges. The basic blocks represent nodes of the graph. The directed edges represent jumps in the control flow. For each control instruction, we insert a *BasicBlock* at the end of the basic blocks list. Then, we specify the target of basic blocks. We perform the post dominator analysis (A node v is post-dominated by a node w in the control flow graph G if every path from v to $Exit$, not including v, contains w) on the control flow graph to determine the flow of the condition dependencies from different blocks. After this, we allocate a *BitmapBits* for tracking condition dependency. We store the control flow graph using the DOT language of graphviz tool [4] in the data directory of the smartphone.

4.2 Dynamic Analysis Component

The dynamic analysis is performed at run time by instrumenting the Dalvik virtual machine interpreter. We assign a *context_taint* to each basic block. The *context_taint* includes the taint of the condition on which the block depends. We compare arguments in the condition using the following instruction : $res\_cmp = ((s4)GET\_REGISTER(vsrc1)\_cmp\ (s4)GET\_REGISTER(vsrc2))$. Based on the comparison result, we verify wether the branch is taken or not. We Combine the taints of different variables of the condition as follows: $SET\_REGISTER\_TAINT(vdst,\ (GET\_REGISTER\_TAINT(vsrc1)|GET\_REGISTER\_TAINT\ (vsrc2)))$ to obtain the $Context\_Taint$. If res_cmp is not null then the branch is not taken. Thus, we adjust the ordinal counter to point to the first instruction of the branch by using the function $ADJUST\_PC(2)$. Otherwise, it is the second branch (else) which is not taken then we adjust the ordinal counter to point to the first instruction in this branch

by using the function $ADJUST\_PC(br)$ where br represents the branch pointer. We instrument different instructions in the interpreter to handle conditional statements. For each instruction, we taint the variable to which we associate a value (destination register). In the case of for and $while$ loops, we process by the same way but we test whether the condition is still true or not in each iteration. We make a special treatment for $Switch$ instructions. We deal with all case statements and all instructions which are defined inside $Switch$ instructions. Note that, we only taint variables and do not modify their values. Once we handle all not taken branches, we restore the ordinal counter to treat the taken branches and we assign taints to modified variables in this branch. We make a special exception handling to avoid leaking information. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block. So, an edge is added in the CFG from the throw statement to the catch block to indicate that the throw statement will transfer control to the appropriate catch block. If an exception occurs, the current context taint and the exception's taint are stored. The variables assigned in any of the catch blocks will be tainted depending on the exception's taint.

5 Evaluation

In this section, we analyse a number of Android applications to test the effectiveness of our approach. Then, we study our taint tracking approach overhead using standard benchmarks. We evaluate the false positives that could occur using our approach. We use a Nexus One mobile device running Android OS version 2.3 enhanced to track implicit flows.

5.1 Effectiveness

To evaluate the effectiveness of our approach, we analyse 27 free Android applications downloaded from the Android Market [1] that manipulated private data. As shown in Table 1, five applications require permissions for contacts and five applications require permissions for camera at install time.

Most of these applications access to locations and phones identity. Also, our analysis showed that these permissions are acquired by the implicit or explicit consent of the user. For example, in the weather application, when the user selects the option "use my location", she gives permission to the application to use and to send this information to the weather server. We found that 14 of these 25 analyzed Android applications (marked with * in the Table 1) leak private information:

- The IMEI numbers that identify a specific cell phone on a network is one of the information that is transmitted by 11 applications. Nine of them do not present an End User License Agreement (EULA).

Table 1. Third party applications grouped by the requested permissions (L: location, Ca: camera, Co: contacts, P: phone state).

Third party applications	Permissions			
	L	Ca	Co	P
The Weather Channel*; Cestos; Solitaire; Babble; Manga Browser (5)	x			
Bump; Traffic Jam; Find It*; Hearts; Blackjack; Alchemy; Horoscope*; Bubble Burst Free; Wisdom Quotes Lite*; Paper Toss*; Classic Simon Free; Astrid* (12)	x			x
Layar*; Knocking*; Coupons*; Trapster*; ProBasketBall (5)	x	x		x
Wertago*; Dastelefonbuch*; RingTones*; Yellow Pages*; Contact Analyser (5)	x		x	x

- Two applications transmitted the device's phone number, the IMSI and the ICC-ID number to their server.
- The location information is leaked by 15 third-party applications to advertisement servers. These applications do not require implicit or explicit user consent. Just two applications require an EULA.

Table 2. Third party applications used control flows

Category	application Name	Leaked Data
Contact and Phone Identity	Wertago	x
	Dastelefonbuch	x
	Yellow Pages	x
Camera	Knocking	x
	ProBasketBall	
Location and Phone Identity	The Weather Channel	x
	Cestos	
	Classic Simon Free	
	Bubble Burst Free	
	Bump	
	Traffic Jam	
	Horoscope	x
	Paper Toss	x
	Find It	x

We use dex2jar tool [2] to translate dex files of different applications to jar files. Then, we use jd-gui [3] to obtain the source code that will

be analysed. As shown in Table 2, we found that 14 of tested Android applications listed by types of accessed sensitive data use control flows to transfer private information. Eight of them leaked private data. Sensitive data is used in the if,for and $while$ control flow instructions. We verify that variables to which a value is assigned in these instructions and that depend on a condition containing private data are not tainted using TaintDroid. Our approach has succesfully propagated taint in these control instructions and detected leakage of tainted sensitive data that is reported in the alert messages.

5.2 Performance

In this part of the paper, we study our taint tracking approach overhead. The static analysis is performed at load and verification time. At load time, our approach adds 33% overhead with respect to the unmodified system. At verification and optimization time, our approach adds 27% overhead with respect to the unmodified system. This time increase is due to the verification of method instructions and the construction of the control flow graphs in the static analysis phase. We install the CaffeineMark application [6] in our Nexus One mobile device to determine the java microbenchmark. Note that the CaffeineMark scores roughly correlate with the number of Java instructions executed per second and do not depend significantly on the amount of memory in the system or on the speed of a computers disk drives or internet connection [6].

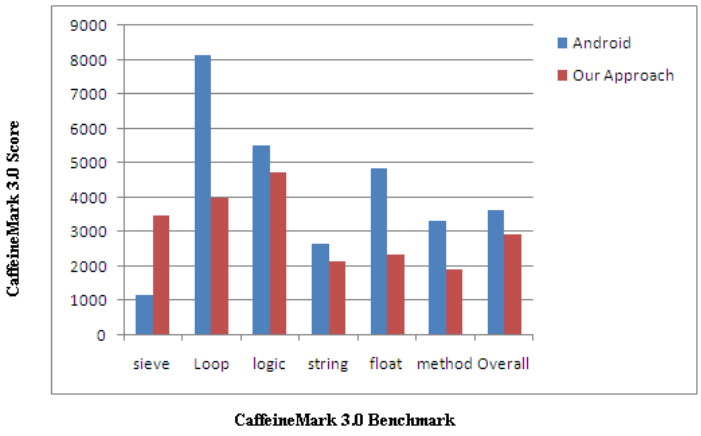

Fig. 4. Microbenchmark of java overhead

Figure 4 presents the execution time results of a Java microbenchmark. We propagate taint in the conditional branches especially in the

loop branches and we add instructions in the processor to solve the under tainting problem. Then, the loop benchmark in our approach presents the greatest overhead. We taint results of arithmetic operations in explicit and control flows. Thus, the arithmetic operations present the greatest overhead. The string benchmark difference between unmodified Android system and our approach is due to the additional memory required in the string objects taint propagation. We observe that the unmodified Android system had an overall score of 3625 Java instructions executed per second. Whereas, our approach had an overall score of 2937 Java instructions executed per second. Therefore, our approach has a 19% overhead with respect to the unmodified system.

5.3 False positives

Our analysis and tests indicated that almost of 50% of studied Android applications use control flows and leak sensitive data. Our approach generates 25% of false positives. We detect an IMSI leakage vulnerability when it is really used as a configuration parameter in the phone. Also, we detect that the IMEI is transmitted outside of smartphone but it is the hash of the leaked IMEI. Thus, we can not treat these applications as privacy violations.

6 Conclusion

In order to detect the leakage of sensitive information by third-party apps exploiting control flows in smartphones, we have proposed a hybrid approach that propagates taint along control dependencies to solve under-tainting problem. We have analysed 27 free Android applications to evaluate the effectiveness of our approach. We found that 14 applications use control flows to transfer sensitive data and 8 leak private information. We showed that our approach generates significant false positives that can be reduced by considering expert rules (ad hoc rules). Also, we can use an access control approach to authorize or not the transmission of the data outside the system. Our approach incurs 19% performance overhead that is due to the propagation of taint in the control flow. To improve performance of our system, we suggest implementing the taint propagation mechanism in Just In Time Compiler (JIT) that provides better performance than the interpreter such as a minimal additional memory usage. By implementing our approach in Android systems, we successfully protect sensitive information and detect most types of software exploits caused by control flows.

References

1. Android, `http://www.android.com/`

2. dex2jar, http://code.google.com/p/dex2jar/
3. Java decompiler, http://jd.benow.ca/
4. AT, Research, T.: Graphviz, http://www.graphviz.org/
5. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. pp. 239–252. ACM (2011)
6. CORPORATION, P.S.: Caffeinemark 3.0, http://www.benchmarkhq.ru/cm30/
7. Denning, D.: Secure information flow in computer systems. Ph.D. thesis, Purdue University (1975)
8. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: Pios: Detecting privacy leaks in ios applications. In: Proceedings of the Network and Distributed System Security Symposium (2011)
9. Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation. pp. 1–6. USENIX Association (2010)
10. Fenton, J.: Memoryless subsystem. Computer Journal 17(2), 143–147 (1974)
11. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications. Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa (2009)
12. Graa, M., Cuppens-Boulahia, N., Cuppens, F., Cavalli, A.: Detecting control flow in smarphones: Combining static and dynamic analyses. In: Cyberspace Safety and Security, pp. 33–47. Springer (2012)
13. Graa, M., Cuppens-Boulahia, N., Cuppens, F., Cavalli, A.: Formal Characterization of Illegal Control Flow in Android System. In: 9th International Conference on Signal Image Technology & Internet Systems (2013)
14. Graa, M., Cuppens-Boulahia, N., Cuppens, F., Cavalli, A.: Protection against Code Obfuscation Attacks based on control dependencies in Android Systems. In: 8th International Workshop on Trustworthy Computing (2014)
15. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 639–652. ACM (2011)
16. Kang, M., McCamant, S., Poosankam, P., Song, D.: Dta++: Dynamic taint analysis with targeted control-flow propagation. In: Proc. of the 18th Annual Network and Distributed System Security Symp. San Diego, CA (2011)
17. Rob van der Meulen, J.R.: Gartner says smartphone sales accounted for 55 percent of overall mobile phone sales in third quarter of 2013 (2013), http://www.gartner.com/newsroom/id/2623415
18. Nair, S., Simpson, P., Crispo, B., Tanenbaum, A.: A virtual machine based information flow control system for policy enforcement. Electronic Notes in Theoretical Computer Science 197(1), 3–16 (2008)
19. News, B.: Bbc google activations and downloads update may 2013 (May 2013), http://www.bbc.com/news/technology-22542725
20. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. Information Systems Security pp. 1–25 (2008)
21. Wilson, T.: Many android apps leaking private information (July 2011), http://www.informationweek.com/security/mobile/many-android-apps-leaking-private-inform/231002162